HiBD
High-Performance Big Data

MVAPICH
MPI, PGAS and Hybrid MPI+PGAS Library

HiDL
High-Performance Deep Learning

Towards Java-based HPC using the MVAPICH2 Library

9th MVAPICH User Group (MUG) Meeting '21

Aamir Shafi

The Ohio State University
shafi.16@osu.edu
https://cse.osu.edu/people/shafi.16

Follow us on

https://twitter.com/mvapich

Presentation Outline

- Introduction
- Design and Implementation of Java Bindings for MVAPICH2
- Performance Evaluation
- Summary

Background

- Standardization efforts for developing Java bindings for MPI:
  - The Java Grande Forum—formed in late 90s—came up with an API called mpiJava 1.2
  - The MPJ API followed that is a minor upgrade to the mpiJava 1.2 API
- Existing Java MPI Libraries:
  - mpiJava: http://www.hpjava.org/mpiJava.html
  - MPJ Express: http://mpjexpress.org/
  - FastMPJ: http://gac.udc.es/~rober/fastmpj
  - Open MPI Java Bindings: https://www.open-mpi.org/faq/?category=java
  - API mismatches between these MPI libraries

Why Java?

- Portability
- A popular language in colleges and software industry:
  - Large pool of software developers
  - A useful educational tool
- Higher programming abstractions including OO features
- One of the largely adopted language by the Big Data community
- Improved compile and runtime checking of the code
- Automatic garbage collection
- Support for multithreading
- Rich collection of support libraries

Introduction

- This effort aims to produce prototype Java bindings for the MVAPICH2 library
  - Initially we plan to roll out support for common MPI functions including:
    - Blocking/non-blocking point-to-point functions
    - Blocking collective functions
    - Strided blocking collective functions
    - Communicator and group management functions
  - Java bindings in the MVAPICH2 library will initially support Open MPI Java bindings with slight modifications
  - Also included is a test-suite to check correctness of Java bindings
- In a parallel effort, we are also adding support for Java micro-benchmarks in Ohio Micro-Benchmark (OMB) suite:
  - Point-to-point, blocking collectives, and strided blocking collectives

Architecture of MVAPICH2 Software Family

High Performance Parallel Programming Models

Java Bindings

Message Passing Interface (MPI)

PGAS (UPC, OpenSHMEM, CAF, UPC++)

Hybrid --- MPI + X (MPI + PGAS + OpenMP/Cilk)

High Performance and Scalable Communication Runtime

Diverse APIs and Mechanisms

- Point-to-point Primitives
- Collectives Algorithms
- Job Startup
- Energy-Awareness
- Remote Memory Access
- I/O and File Systems
- Fault Tolerance
- Virtualization
- Active Messages
- Introspection & Analysis

Support for Modern Networking Technology (InfiniBand, iWARP, RoCE, Omni-Path)

Transport Protocols: RC, XRC, UD, DC

Modern Features: SHARP2*, ODP, SR-IOV, Multi Rail

Support for Modern Multi-/Many-core Architectures (Intel-Xeon, OpenPower, Xeon-Phi, ARM, NVIDIA GPGPU)

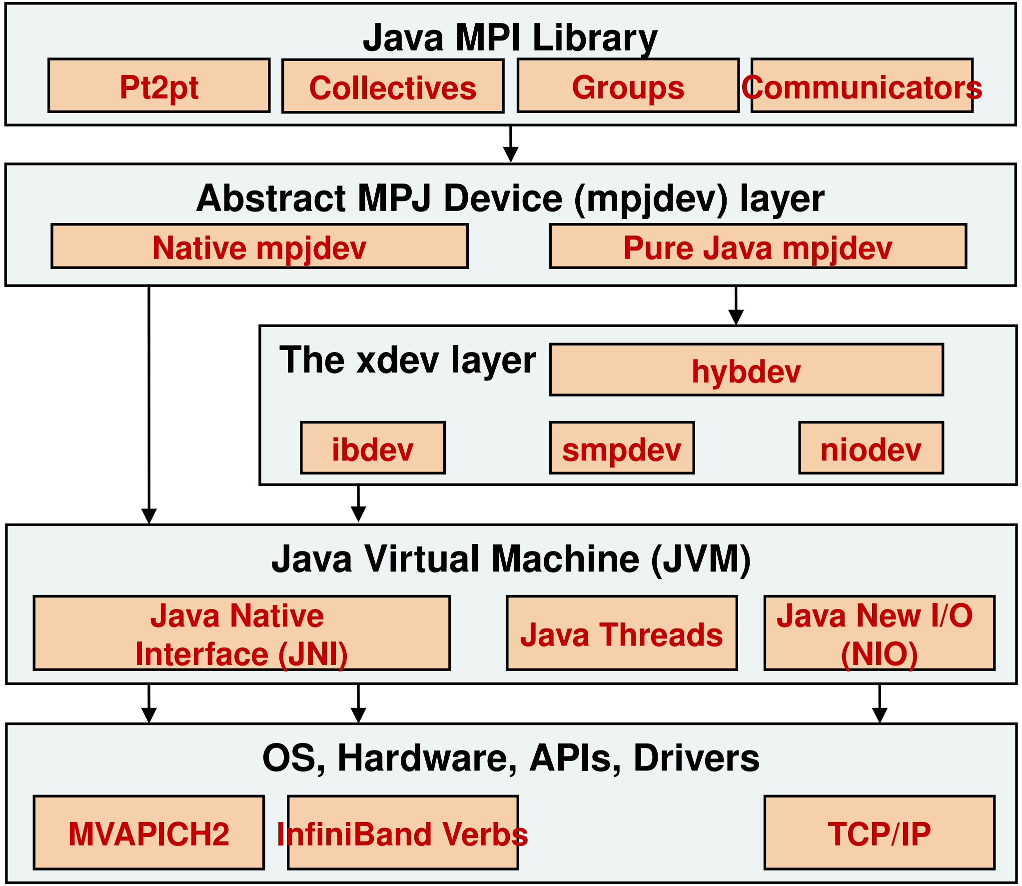
Transport Mechanisms: Shared Memory, CMA, IVSHMEM, XPMEM

Modern Features: MCDRAM*, NVLink, CAPI*

* Upcoming

Earlier Approaches

- Design adopted by an earlier Java MPI library (MPJ Express)
- Makes it easier to support new interconnects
- But higher-level MPI concepts are fully implemented in Java
- Pure Java communication devices exhibit poor performance
- Based on this, the current Java bindings aim to keep Java layer "as minimal as possible"

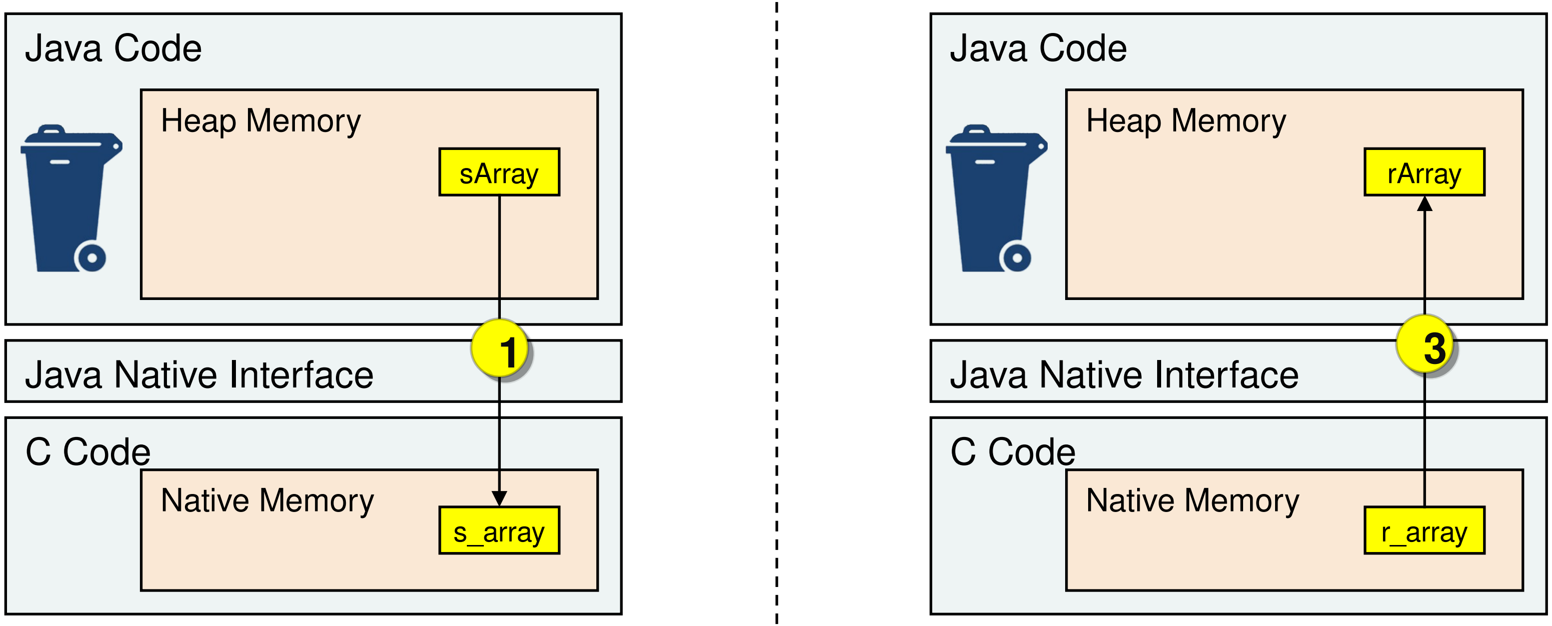


Implementation: Data movement for Blocking send()/recv()

Sender | **Receiver**

MPI.COMM_WORLD.Send(sArray, 4, MPI.INT, dest, tag)

MPI.COMM_WORLD.Recv(rArray, 4, MPI.INT, dest, tag



MPI_Send(s_array, 4, MPI_INT, dest, tag, ..) —— 2 ——> MPI_Recv(r_rarray, 4, MPI_INT, dest, tag, ..)

Implementation: Passing Data from Java to C code

Application code

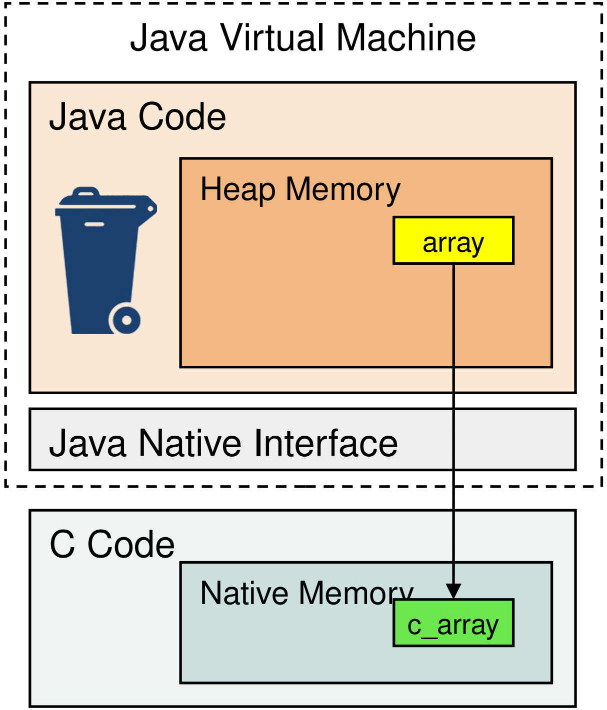
```
int[] array = new int[100];
MPI.COMM_WORLD.send(array, 100, MPI.INT, dst, tag);
```

Java send() code

```
//Blocking Comm.send() method
void send(Object buf … ) {
    nativeSend(buf …);
}
```

C send() code invoked from Java using JNI

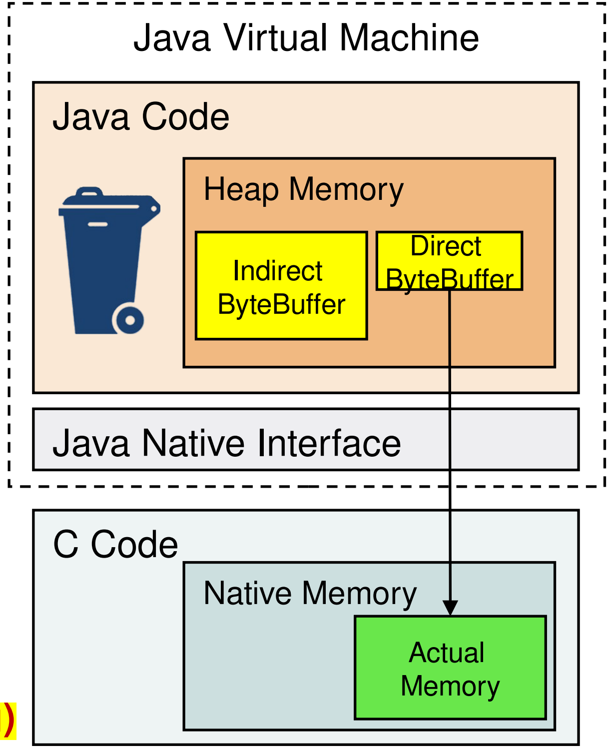
```
//C implementation of nativeSend
JNIEXPORT void JNICALL nativeSend(JNIEnv *env,
                                  jobject buf … )
{
    void *bufptr = (void *)(*env)->GetIntArrayElements( buf …) ;
    MPI_Send(bufptr …) ;
    (*env)->ReleaseByteArrayElements( .. bufptr .. );
```



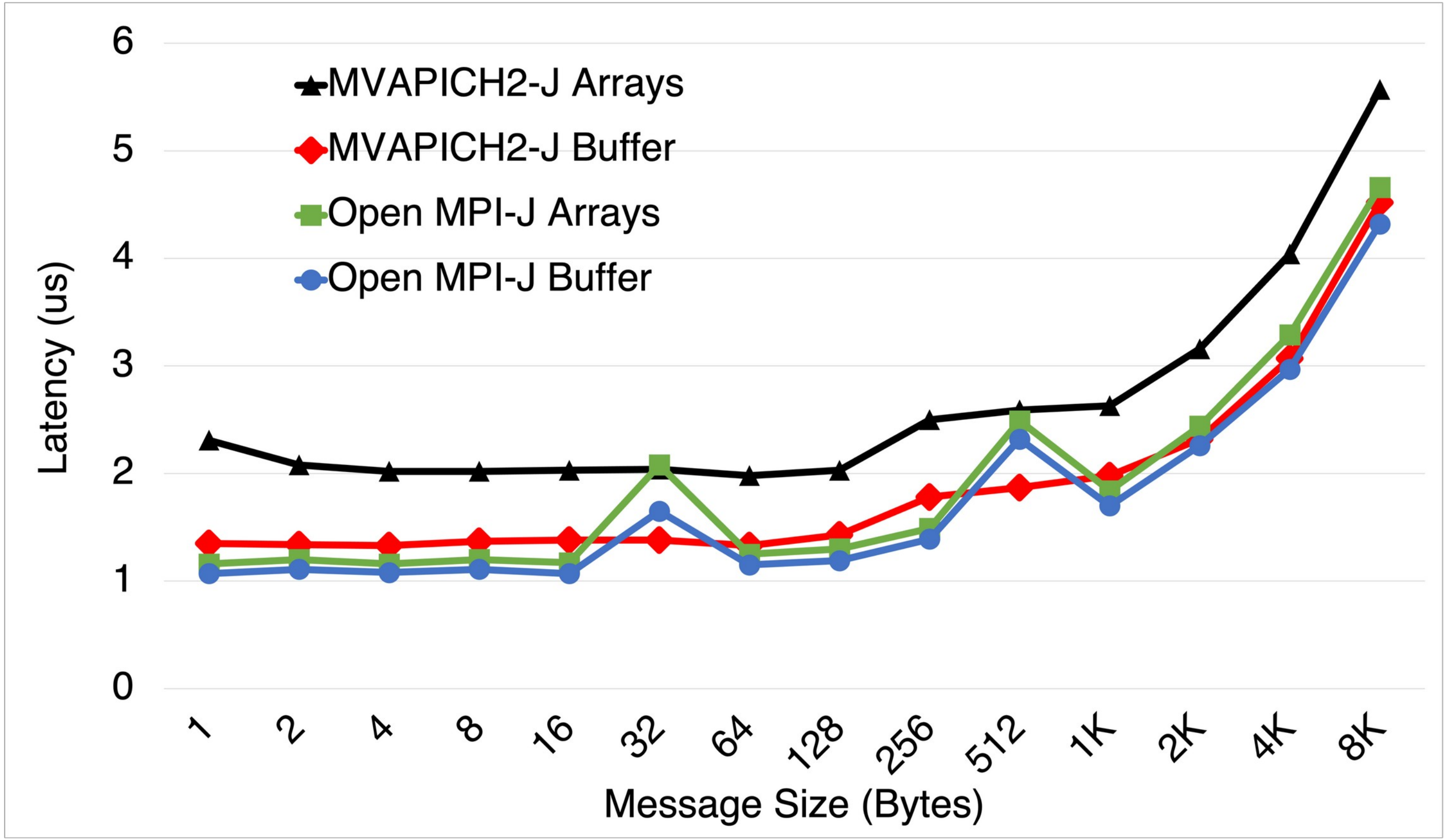
Implementation: Our Approach (Direct ByteBuffers)

- MVAPICH2 Java bindings support communication to/from:
  - Java arrays :
    - There is 1 extra data copy at sender and receiver each!
  - Direct ByteBuffers
- For Java arrays our implementation uses a memory management library based on the direct ByteBuffers (inspired from MPJ Express):
    - Key takeaway - It is possible to retrieve pointer to direct ByteBuffer as these are not subject to garbage collection
- To tackle this extra copy, our bindings support exchanging data from direct ByteBuffers

MPI.COMM_WORLD.Send(directByteBuffer, 4, MPI.BYTE, dest, tag)



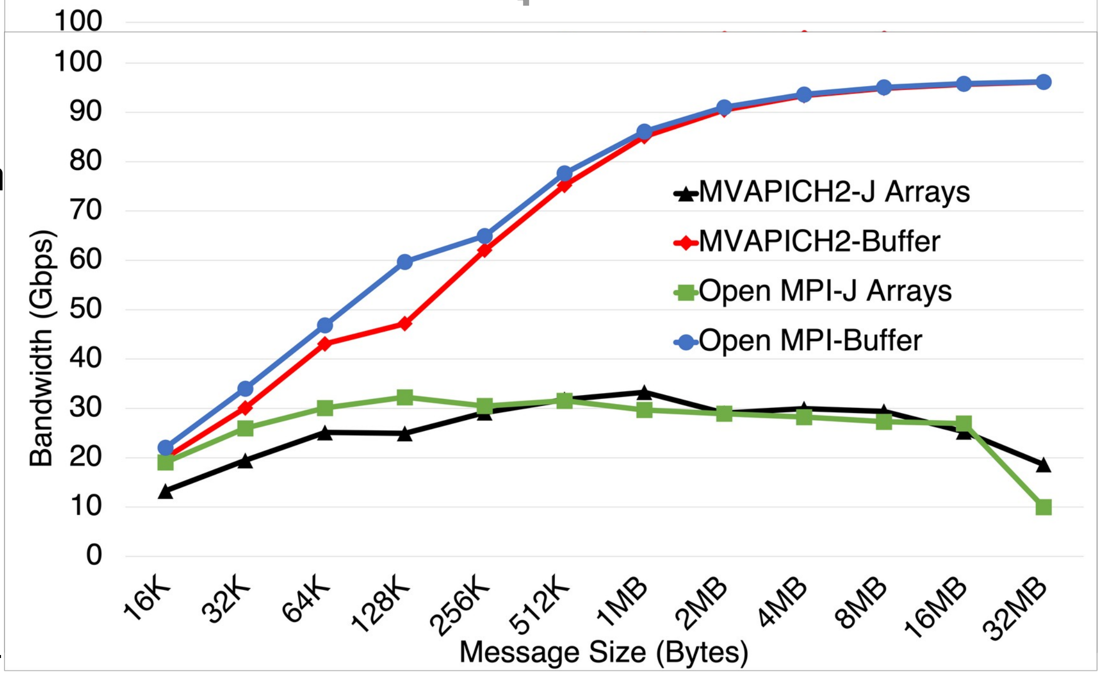
Preliminary Latency Comparison



Preliminary Bandwidth Comparison

- The bandwidth graph for Buffer vs. Arrays show that arrays are slower due to an additional data copy
- It is not possible to acquire bandwidth numbers with Open MPI Java bindings because it does not support communicating Java arrays with non-blocking send/recv methods:
  - Non-blocking methods are used by OSUBandwidth benchmark to measure bandwidth
  - Communicating Java arrays with non-blocking MPI methods has been part of all Java MPI libraries and APIs

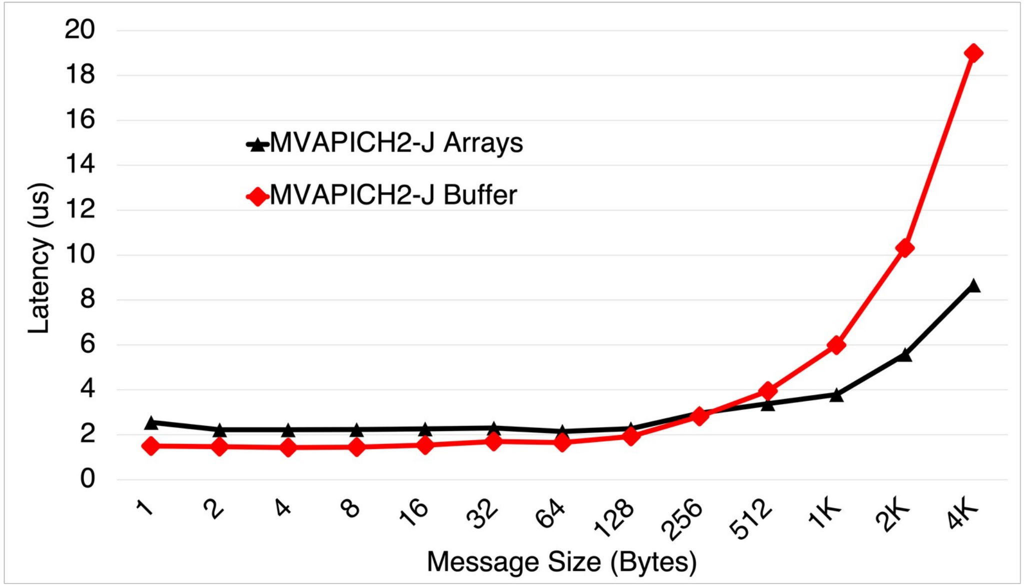
Java arrays perform much slower due to extra copies!



Should we just use ByteBuffers in applications?

Preliminary Latency Comparison with Data Validation

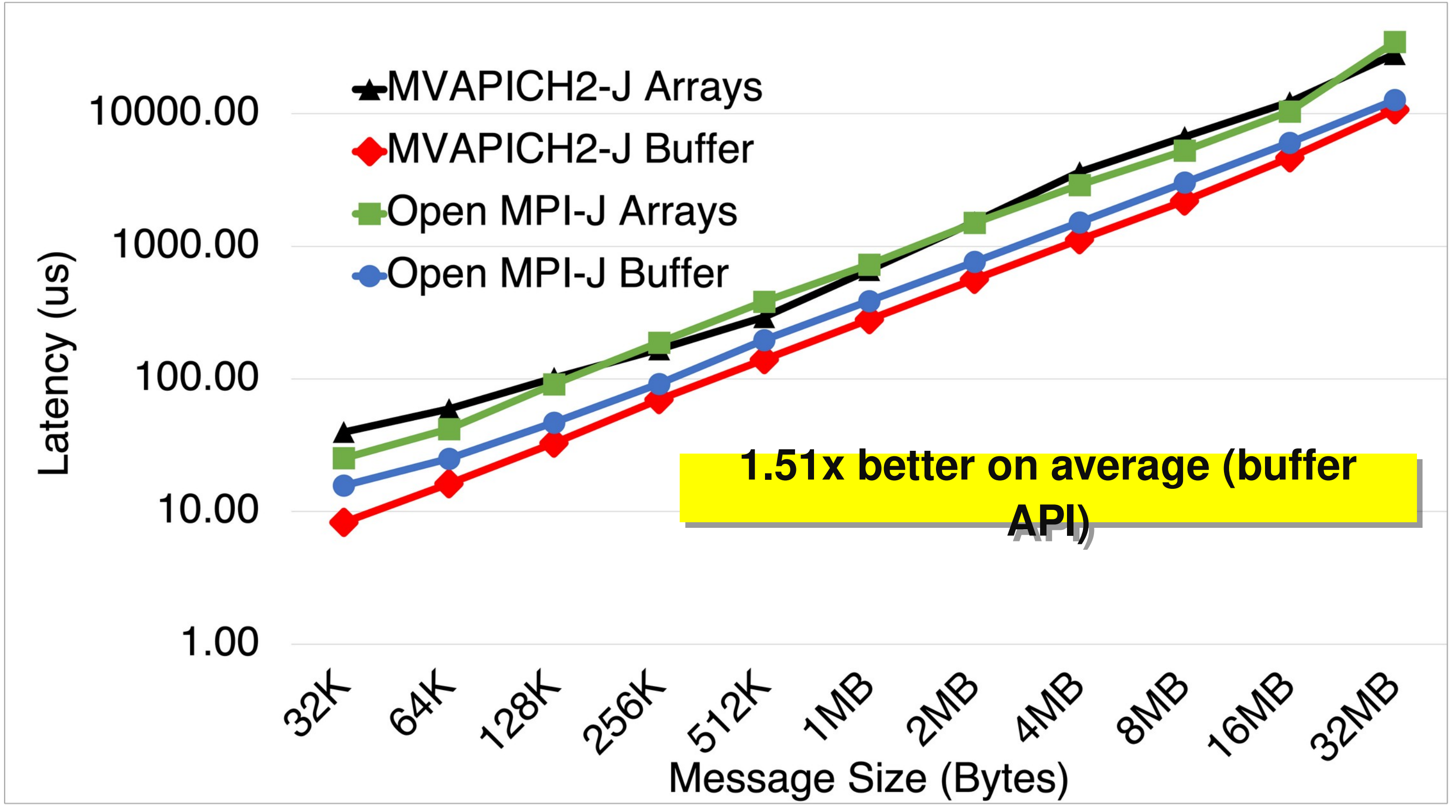
- Is it beneficial to use ByteBuffers as compared to Java arrays?
  - Will force applications to use (and store data) in ByteBuffers
  - However, while communicating ByteBuffers is faster than arrays, reading/writing data from ByteBuffer is slower than arrays
- Latency comparison with data validation (used as dummy compute)




```
boolean validateDataAfterRecv(byte[] src, byte[] dst, int count) {
  for(int i=0 ; i<count ; i++) {
    if(src[i] != dst[i])
      return false;
  }
  return true;
}
```

```
boolean validateDataAfterRecv(ByteBuffer src, ByteBuffer dst, int count) {
    for(int i=0 ; i<count ; i++) {
      if(src.get() != dst.get())
        return false;
    }
    return true;
}
```

Preliminary Bcast Comparison – 8 processes



Summary

- The talk presented early experiences of implementing Java bindings for MVAPICH2:
  - Relies on a memory management layer that exploits direct ByteBuffers
  - Supported features:
    - Blocking/non-blocking point-to-point functions
    - Blocking collective functions
    - Strided blocking collective functions
    - Communicator and group management functions
- Future work:
  - Continue further development of Java bindings
  - Evaluate performance using benchmarks (NPB) and real-world applications
  - Release Java bindings and Java OMB

Thank You!

shafi.16@osu.edu

Network Based Computing Laboratory

Follow us on

https://twitter.com/mvapich

Network-Based Computing Laboratory

http://nowlab.cse.ohio-state.edu/

MVAPICH
MPI, PGAS and Hybrid MPI+PGAS Library

The High-Performance MPI/PGAS Project
http://mvapich.cse.ohio-state.edu/

HiBD
High-Performance Big Data

The High-Performance Big Data Project
http://hibd.cse.ohio-state.edu/

HiDL
High-Performance Deep Learning

The High-Performance Deep Learning Project
http://hidl.cse.ohio-state.edu/

Implementation: Passing Data from Java to C code

- In order copy data (primitive datatype arrays) from Java to C code, the JNI API provides:
  - **Method 1:** Get<*Type*>ArrayElements and Release<*Type*>ArrayElements routines:
    - <*Type*> are primitive Java datatypes like int, byte, float etc.
  - **Method 2:** GetPrimitiveArrayCritical and ReleasePrimitiveArrayCritical routines
- Most Java Virtual Machines (JVMs) today do not support “pinning”:
  - Hence passing data from Java to C incurs a true data copy